Ann Oper Res (2011) 184: 3–26
DOI 10.1007/s10479-010-0730-1

Rapidly computing robust minimum capacity s-t cuts: a case study in solving a sequence of maximum flow problems

Douglas S. Altner · Özlem Ergun

Published online: 23 March 2010

Abstract The Minimum Capacity *s-t* Cut Problem (MinCut) is an intensively studied problem in combinatorial optimization. A natural extension is the problem of choosing a minimum capacity *s-t* cut when arc capacities are unknown but confined to known intervals. This motivates the Robust Minimum Capacity *s-t* Cut Problem (RobuCut), which has applications such as open-pit mining and project scheduling. In this paper, we show how RobuCut can be reduced to solving a sequence of maximum flow problems and provide an efficient algorithm for rapidly solving this sequence of problems. We demonstrate that our algorithm solves instances of RobuCut in seconds that would require hours if a standard maximum flow solver is iteratively used as a black-box subroutine.

Keywords Maximum flows · Robust minimum cuts · Reoptimization heuristics · Goldberg-Tarjan algorithm · Robust network optimization · Incremental maximum flow algorithms

1 Introduction

The Minimum Capacity *s-t* Cut Problem (MinCut) is a fundamental problem in combinatorial optimization. It has numerous nontrivial applications to a wide selection of real-world problems. For an extensive list of applications, please see Sect. 6.2 of Ahuja et al. (1993). A natural extension of MinCut is the problem of conservatively choosing a *s-t* cut in light of uncertain arc capacities. Specifically, we propose and study the Robust Minimum Capacity *s-t* Cut Problem (RobuCut) where arc capacities are unknown but confined to known intervals.

D.S. Altner (✉)
Department of Mathematics, United States Naval Academy, 572C Holloway Road, Annapolis, MD 21402, USA
e-mail: altner@usna.edu

Ö. Ergun
H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, 765 Ferst Drive NW, Atlanta, GA 30332, USA
e-mail: oergun@isye.gatech.edu

Robust programming is a branch of mathematical programming that models conservative planning under data uncertainty. Intuitively, robust programming allows a user to maximize his profit or minimize his costs in a "bad" scenario. Since planning for the worst possible scenario is often too conservative, robust programming includes a *parameter of robustness* that allows the decision maker to specify a desired degree of conservative planning. In this paper, we use the polyhedral model of uncertainty of Bertsimas and Sim (2003).

RobuCut can be used for several applications of MinCut where arc capacities may be obtained from imprecise engineering estimates. For example, in the application of open-pit mining, a minimum capacity s-t cut (minimum cut) in the formulated network indicates which blocks of earth should be excavated to maximize profit subject to constraints on the slope requirement of the pit and precedence constraints on the blocks (Hochbaum and Chen 2000). In this model, the arc capacities correspond to the estimated economic value of the ore in each block, which relies on geological information obtained from drill cores and the forecasting of future commodity prices. Due to the number of critical assumptions that serve as the basis for widely used open-pit mining decision models, industry experts have emphasized the need for models that incorporate the uncertainty of the economic value of the orebodies (Monkhouse and Yeates 2005). One such method of encapsulating this uncertainty is to model the open-pit mining problem as a robust optimization problem.

There are several other applications of MinCut that can be extended to a robust programming framework. Stone (1977) proposes a MinCut model that can be used for distributed computing on a dual-processor machine with the objective of minimizing the amount of communication required between the two processors. With RobuCut, this model can be extended to ensure the computer user never has to wait "too" long even in a "bad" case. Möhring et al. (2003) show that a cost minimization scheduling problem with start-time dependent costs can be formulated with MinCut. With RobuCut, this model can be extended to incorporate the uncertainty in cost that could stem from unplanned delays. For example, in the context of construction management, these could be delays caused by weather or accidents. Harris and Ross (1955) discuss how to model finding a minimum cost interdiction of a s-t flow network as a MinCut. With RobuCut, their model can be extended to incorporate uncertainty of the costs associated with destroying arcs. For example, in the context of interdicting a railroad network, this allows one to hedge against several real-world uncertainties such as the strength of defenses as well as the success of the air strikes.

Bertsimas and Sim (2003) initiate the study of robust combinatorial optimization and network flows. In addition to providing a modeling framework, the authors also prove that any robust combinatorial optimization problem (RobuCOP) can be solved by solving a linear number of nominal combinatorial optimization problems. Thus, RobuCut may be solved by solving a linear number of MinCuts.

A fundamental result of network optimization is that a minimum cut may be constructed by computing a maximum s-t flow (maximum flow). Initially, it may seem as if no further study is required. After all, a RobuCut can be solved with at most a linear number of maximum flow computations and there are quite effective open-source solvers for the maximum flow problem that are freely available on the Internet (Goldberg 2010). However, we show that using a maximum flow solver as a black-box subroutine when solving RobuCut can lead to hours of unnecessary computations. As a remedy, we propose an algorithm that relies on rapid maximum flow reoptimization heuristics.

Given that RobuCut can be reduced to solving a sequence of maximum flow problems, our study also serves as a case study of the effectiveness of maximum flow reoptimization heuristics. Large sequences of similar maximum flow problems arise in a diverse collection of practical solution techniques for complex, real-world problems including those in

computational biology (Strickland et al. 2005), constraint programming (Régin 1994) and biometry (Govindaraju 2008). For an extensive list of algorithms that involve solving a large sequence of maximum flow problems as well as a detailed study of maximum flow reoptimization heuristics, see Altner (2008).

RobuCut is also one of several important problems in the burgeoning collection of research literature on robust network programming. We briefly survey a few other papers in robust network programming. Chaerani and Roos (2006) formulate a robust maximum flow problem as a conic program using the ellipsoidal model of uncertainty of Ben-Tal and Nemirovski (1998). Atamtürk and Zhang (2007) develop a two-stage robust optimization approach for solving network flow and design problems with uncertain demand. The authors generalize the approach to multicommodity flow network design and suggest applications to lot-sizing and location-transportation problems. Ordóñez and Zhao (2007) develop a robust programming formulation for the problem of expanding arc capacities in a network subject to demand and travel uncertainty. The authors also prove their model can be reformulated as a conic linear program.

The main contributions of this paper are developing an efficient algorithm for RobuCut and demonstrating that our algorithm drastically outperforms two other algorithmic approaches: one that uses a maximum flow solver as a black-box subroutine and another that incrementally uses a maximum flow simplex algorithm as a subroutine. Specifically, we demonstrate our algorithm solves RobuCut instances with hundreds of nodes in seconds whereas the other algorithms could require more than four hours. Thus, we have turned what could require half of a working day's worth of time into a near real-time decision.

In Sect. 2, we discuss network flow preliminaries. In Sect. 3, we formally introduce the Robust Minimum s-t Cut Problem and prove a worst-case algorithmic result. In Sect. 4, we provide a detailed description of our algorithm and prove it is no worse, in terms of worst-case analysis, than making a linear number of subroutine calls to the highest label pre-flow push algorithm of Goldberg and Tarjan (1988). In Sect. 5, we detail our experiments and present computational results. In Sect. 6, we draw conclusions.

2 Preliminaries

2.1 Notation

We use $N = (V, A)$ to denote a network or directed graph with node set V and arc set A. An arc from node i to node j is denoted as (i, j). $x_{i,j}$ and $u_{i,j}$ are used to denote the flow on and the capacity of arc (i, j), respectively. Every network mentioned in this paper is a single commodity flow network and has a unique source $s \in V$ and a unique sink $t \in V$. We assume there are no arcs entering s and there are no arcs leaving t. All problems discussed in this paper are defined on networks that are s-t connected; that is, there exists a directed path from node s to node t. If it is possible to send at least one unit of flow from a node u to a node v then node v is *reachable* from node u.

Given a node v, the *forward star* of v, that is, the set of all arcs leaving v, is denoted by $FS(v)$. Similarly, the set of all arcs entering v is known as the *reverse star* and is denoted by $RS(v)$.

2.2 Network flow preliminaries

For background in network flows, we recommend Ahuja et al. (1993). The reader should be familiar with the *Maximum Flow Minimum Cut Theorem*, which states that the maximum

flow in a s-t network equals the minimum capacity s-t cut in that network. This theorem was originally proved in Ford and Fulkerson (1956).

Given a feasible s-t flow x in a network $N = (V, A)$, we can construct the *residual network* as follows. For each node $v \in V$ we create a corresponding node in the residual network. For each arc $e = (u, v) \in A$ such that $u_e - x_e > 0$ we create a corresponding arc $e_f = (u, v)$ in the residual network with capacity $u_{e_f} = u_e - x_e$. Similarly, for each arc $e = (u, v) \in A$ such that $x_e > 0$ we create a corresponding arc $e_b = (v, u)$ in the residual network with capacity $u_{e_b} = x_e$. The source and sink in the residual network correspond to the source and sink respectively of the original network. The following result is well known:

Theorem 1 *A feasible s-t flow is a maximum flow in a network N if and only if the corresponding residual network has a maximum flow of* 0.

2.2.1 Goldberg-Tarjan algorithm

We refer to the well known maximum flow algorithm of Goldberg and Tarjan (1988) as the Goldberg-Tarjan algorithm. This algorithm is also known as the pre-flow push algorithm and as the push-relabel algorithm.

At any point during the Goldberg-Tarjan algorithm, every node v has an associated *distance label* $d(v)$ and an *excess* $e(v)$. The distance label is a lower bound on the shortest distance, in terms of the number of arcs, from v to t. Upon initiation, we set $d(s) = |V|$ and $d(t) = 0$. The excess of a node v is defined as $e(v) = \sum_{i \in RS(v)} x_{(i,v)} - \sum_{j \in FS(v)} x_{(v,j)}$. Any node with a positive excess is an *active node*.

We say that a residual arc (u, v) is *admissible* if and only if $d(u) = d(v) + 1$. Throughout the course of the Goldberg-Tarjan algorithm, admissible arcs are the only arcs that have their current value of flow adjusted.

A *pseudoflow* is a flow that satisfies arc bounds but does not necessarily satisfy the flow balance constraints. A *pre-flow* is a pseudoflow where the flow entering a node is always greater than or equal to the flow leaving a node. We refer to the quantity $\sum_{i \in V: e(v) > 0} e(v)$ as the *amount of pre-flow* in a network.

The pseudocode for the Goldberg-Tarjan algorithm is contained in Algorithm 1.

Initialize $d(v)$ and $e(v)$ $\forall$ $v \in V$
$x_e \leftarrow u_e$ $\forall$ $e \in FS(s)$
$x_e \leftarrow 0$ $\forall$ $e \notin FS(s)$

while There is an active node i **do**
  if the residual network contains an admissible arc (i, j) **then**
    Push $\delta := \min\{e(i), u_{(i,j)} - x_{(i,j)}\}$ units of flow from node i to node j
  else
    $d(i) \leftarrow \min\{d(j) + 1 : (i, j) \in \text{residual } FS(i)\}$
  end if
end while

Algorithm 1: Goldberg-Tarjan algorithm

The Goldberg-Tarjan algorithm maintains a pre-flow as an invariant and strives to convert the pre-flow into a maximum flow. At the beginning of each iteration, we find an active

node i. If there is no active node, then we terminate with a maximum flow. Otherwise, we find an admissible arc in $FS(i)$ in the residual network and augment its flow. If no such admissible arc exists, then we *relabel* node i. The step: $d(i) \leftarrow \min \{d(j) + 1 : (i, j) \in \text{residual } FS(i)\}$ denotes relabeling.

The success of the Goldberg-Tarjan algorithm was partly attributed to the implementation of both gap relabeling and global relabeling heuristics. These heuristics are detailed in Cherkassky and Goldberg (1994) and their discussion is beyond the scope of this paper.

In terms of implementation, we implemented the Highest-Label Goldberg-Tarjan algorithm. That is, we always choose the active node with the highest distance label for the discharge operation. This is regarded as the fastest implementation of the Goldberg-Tarjan algorithm in practice (Cherkassky and Goldberg 1994). We also implemented both the global and the gap relabeling heuristics.

2.3 Storing minimum capacity s-t cuts

Given a network and a corresponding maximum flow, we can use the following data structure to store two important minimum cuts:

Definition 1 Given a network that is currently at maximum flow, a *cut tripartition* is a tripartition $(V_s, V \backslash (V_s \cup V_t), V_t)$ of the node set V according to the following schema: V_s is the set of all nodes currently reachable from the source in the optimal residual network. V_t is the set of all nodes that can currently reach the sink in the optimal residual network.

A cut tripartition implicitly stores two, not necessarily unique, minimum cuts: $C_s = (V_s, V \backslash V_s)$ and $C_t = (V \backslash V_t, V_t)$. We alternatively denote a cut tripartition as $\{C_s, C_t\}$.

Once it is stored, a cut tripartition can indicate if an arc is contained in all minimum cuts in constant time. This is because an arc e is contained in all minimum cuts if and only if $e \in C_s \cap C_t$.

A cut tripartition is used later in the paper to obtain a good upper bound on the new maximum flow value in a network after a few arc capacities have been increased. Altner and Ergun (2008) first introduced the cut tripartition.

3 The robust minimum capacity s-t cut problem

In the first subsection in this section, we formally introduce the Robust Minimum Capacity s-t Cut Problem. In the second subsection, we present a worst-case algorithmic result.

3.1 Problem statement

Robust Minimum Capacity s-t Cut Problem: Let $N = (V, A)$ be a network with source s and sink t. Assume arc capacities $\tilde{u_e}$ are unknown but are known to take value in $[u_e, u_e + d_e]$ for all $e \in A$. Choose a s-t cut C that is of minimax capacity given that Γ arcs will assume their highest possible capacity after C is selected and all other arcs will assume their lowest possible capacity.

We may assume that the arcs are enumerated $A = \{e_0, e_1, \ldots, e_{|A|-1}\}$ such that $d_{e_0} \geq d_{e_1} \geq \cdots \geq d_{e_{|A|-1}}$. For notational convenience, we define $d_{e_{|A|}}$ to be 0. In the robust optimization literature, Γ is referred to as the *robust parameter of optimization.* The user assigns Γ an integer value from the interval $[0, |A|]$ to quantify his desired degree of conservative

planning. Let ζ be the family of all s-t cuts in the network N. The Robust Minimum Capacity s-t Cut Problem (RobuCut) can be formally written as follows:

$$\begin{aligned} &\text{Minimize} \quad \sum_{e\in C} u_e + \max_{\{S|S\subseteq A,|S|\leq \Gamma\}} \sum_{j\in S\cap C} d_j \\ &\text{Subject to} \quad C \in \zeta \end{aligned}$$

Theorem 2 *RobuCut may be solved by computing $|A|+1$ minimum cuts. Specifically, by solving the following optimization problem*:

$$Z^* = \min_{\ell=0,\ldots,|A|} G^\ell$$

where for $\ell = 0,\ldots,|A|$:

$$G^\ell := \Gamma d_{e_\ell} + \min_{C\in\zeta}\left\{\sum_{e\in C} u_e + \sum_{\{e_j\in C: j\leq \ell\}} (d_{e_j} - d_{e_\ell})\right\}$$

Proof Immediate corollary of Theorem 3 in Bertsimas and Sim (2003). □

Corollary 1 *RobuCut may be solved by computing $|A|+1$ maximum flows.*

Proof This follows immediately from the previous theorem and by the Maximum Flow Minimum Cut Theorem, which was originally proved in Ford and Fulkerson (1956). □

The sequence of maximum flow problems will henceforth be referred to as the sequence of *nominal maximum flow problems*.

3.2 Algorithmic result

Theorem 3 *Consider an instance of RobuCut on a network $N=(V,A)$. This problem may be solved in $O(G_r + |C^*_{card}|\ |A|\ d_{e_0})$ time where $G_r = \min(|V|^{\frac{2}{3}}, \sqrt{|A|})\ |A|\log(\frac{|V|^2}{|A|}) \times \log(u_{max})$, C^*_{card} is a minimum cardinality s-t cut in N and $u_{max} = \max\{u_e|e\in A\}$.*

Proof Goldberg and Rao (1998) demonstrate that a maximum flow in a network may be computed in $O(G_r)$, where G_r is defined as above. To obtain the desired algorithmic bound, we describe an algorithm for solving the sequence of nominal maximum flow problems. We then use the optimal objective values of these nominal problems to construct an optimal solution to our instance of RobuCut. We may assume that the 0th nominal maximum flow problem is solved by the algorithm of Goldberg and Rao.

Consider the ith nominal maximum flow problem. Note this problem differs from the $(i-1)$st nominal maximum flow problem in that in the capacities of arcs $e_0, e_1, \ldots, e_i$ are increased by $d_{e_{i-1}} - d_{e_i}$. Let C^*_{card} denote a minimum cardinality s-t cut in N. Then the inequality $z^*_i - z^*_{i-1} \leq |C^*_{card}|(d_{e_{i-1}} - d_{e_i})$ holds true, where z^*_i denotes the optimal objective value of the ith nominal maximum flow problem.

We can solve the ith nominal maximum flow problem using an augmenting path algorithm with an optimal solution from the $(i-1)$st nominal maximum flow problem as an initial solution. Note $|C^*_{card}|(d_{e_{i-1}} - d_{e_i})$ is an upper bound on the maximum number of augmenting paths that must be found in the corresponding residual network until a maximum

flow for the ith nominal problem is obtained. Since an augmenting path can be found in at most $O(|A|)$, we conclude we can compute the maximum flow value of the ith nominal maximum flow problem in $O(|C^*_{card}|(d_{e_{i-1}} - d_{e_i})|A|)$.

Since we must solve $|A|$ nominal maximum flow problems after the 0th, the total number of computations to compute all of the subsequent maximum flow values may be bounded above by $\sum_{i=1}^{|A|} |C^*_{card}|(d_{e_{i-1}} - d_{e_i})|A| = |C^*_{card}|d_{e_0}|A|$.

What remains to be addressed is how to construct an optimal objective value to an instance of RobuCut given the optimal objective values to each of the nominal problems: $z^*_0, z^*_1, \ldots, z^*_{|A|}$. This can be constructed by solving the following optimization problem:

$$\min_{\ell \in \{0,\ldots,|A|\}} \{\Gamma d_{e_\ell} + z^*_\ell\}$$

which can be solved in $O(|A|)$ time. The corresponding optimal solution can be obtained similarly. The desired result follows. □

4 Algorithm for RobuCut

This section focuses on our algorithmic approach to solving RobuCut. In the first subsection, we briefly overview our algorithm for RobuCut. In the second subsection, we identify properties of the sequence of maximum flow problems that stem from Corollary 1. In the third subsection, we discuss a heuristic for solving this sequence of maximum flow problems. In the fourth subsection, we provide a detailed description of our algorithm. In the third subsection, we prove that our algorithm is no worse, in terms of worst-case analysis, than iteratively using a black-box highest-label Goldberg-Tarjan algorithm $|A| + 1$ times.

4.1 Overview of algorithm

In this subsection, we provide a broad overview of our algorithm for RobuCut.

Recall that Bertsimas and Sim (2003) provide a general algorithm for solving any RobuCOP, which they refer to as Algorithm A. Algorithm A consists of solving a linear number of nominal COPs, which is a minimum cut problem in the case of RobuCut. To allow for the use of maximum flow reoptimization heuristics, we take the dual of each of these minimum cut problems.

Our algorithm for RobuCut will solve two sequences of maximum flow problems in an alternating fashion. The first sequence is a sequence of auxiliary maximum flow problems on "incremental networks." Solving these problems improves the running time of our algorithm and incremental networks are precisely defined in Sect. 4.3. The second sequence of maximum flow problems is the $|A| + 1$ nominal maximum flow problems. The networks that underlie the nominal maximum flow problems are called the *nominal networks*.

At the beginning of the ith iteration, we assume we have both a maximum flow in the $(i-1)$st incremental network as well as a maximum flow in the $(i-1)$st nominal network. During the ith iteration, we perform the following computations:

1. Use the maximum flow in the $(i-1)$st incremental network to as a starting solution for computing the maximum flow in the ith incremental network. Since the incremental network is a unit capacity network and the ith incremental network contains all of the arcs in the $(i-1)$st incremental network plus one additional arc, this step requires at most the computation of a single augmenting path.

2. Use both the maximum flow in the ith incremental network and the maximum flow in the $(i-1)$st incremental network to get an initial feasible flow for the maximum flow problem on the ith nominal network.
3. Use a modified Goldberg-Tarjan algorithm to compute a maximum flow in the ith nominal network using the initial feasible flow computed in the second step. This step involves computing a crude upper bound on the maximum flow to restrict the amount of pre-flow added to the network.

4.2 Properties of the sequence of maximum flow problems

In this subsection, we discuss properties of the sequence of maximum flow problems that must be solved. These properties are employed to develop reoptimization heuristics that improve the running time of our algorithm for RobuCut.

Let $N_0 = (V, A_0), N_1 = (V, A_1), \ldots, N_{|A|} = (V, A_{|A|})$ be the sequence of nominal networks. For each possible i, let u_e^i be the capacity of arc e in network N_i. All of the networks have the same set of arcs although their capacities monotonically increase with i. Let $\hat{A}_i = \{e \in A : u_e^{i-1} < u_e^i\}$. We make the following observations about this sequence of maximum flow problems:

1. $A_i = A_{i+1} \quad \forall\, i \in \{0, \ldots, |A| - 1\}$
2. $u_e^i \leq u_e^{i+1} \quad \forall\, i \in \{0, \ldots, |A| - 1\},\ \forall\, e \in A$
3. $\hat{A}_i \subseteq \hat{A}_{i+1} \quad \forall\, i \in \{0, \ldots, |A| - 1\}$
4. $|\hat{A_{i+1}} \backslash \hat{A}_i| = 1 \quad \forall\, i \in \{0, \ldots, |A| - 1\}$
5. $u_{e_j}^i - u_{e_j}^{i-1} = d_{e_{i-1}} - d_{e_i} \quad \forall\, e_j \in \hat{A}_i,\ \forall\, i \in \{1, \ldots, |A|\}$

The first listed observation indicates that all of the nominal networks have the same set of arcs, even though the capacities on the arcs may differ between nominal networks. The second observation indicates that, for each arc e, the capacity of arc e in the ith nominal network is less than or equal to the capacity of arc e in the $(i+1)$st nominal network. The third observation indicates that the set of arcs whose capacity in the ith nominal network is strictly greater than the arc's corresponding capacity in the $(i-1)$st nominal network is a subset of the set of arcs whose capacity in the $(i+1)$st nominal network is strictly greater than the arc's corresponding capacity in the ith nominal network. The fourth observation indicates there is exactly one arc that is in the latter set but not the former set. The last observation indicates that the difference between each arc's capacity in the ith nominal network and the $(i-1)$st nominal network is either 0 or $d_{e_{i-1}} - d_{e_i}$, where d_{e_i} is the size of the interval that the ith arc's capacity is confined to.

4.3 Heuristic for maximum flow reoptimization

In this subsection, we use the properties of the sequence of maximum flow problems that were identified in the previous subsection to develop a heuristic speedup for solving them. This heuristic speedup is an integral part of our algorithm for RobuCut.

Suppose we want to compute the maximum flow in N_i and that we know that x^{i-1} is a maximum flow in N_{i-1}. Note that x^{i-1} is always a feasible flow in N_i for all possible i. Moreover, suppose there exists an s-t path $P \subseteq \hat{A}_i$. Then we know a priori that the flow of x^{i-1} with $d_{e_{i-1}} - d_{e_i}$ units of flow augmented along path P always routes at least as much flow through N_i as x^{i-1}. Furthermore, we know the flow of x^{j-1} with $d_{e_{j-1}} - d_{e_j}$ units of flow augmented along path P always routes at least as much flow through N_j as x^{j-1}

for each $j \in \{i+1, i+2, \ldots, |A|\}$. Further still, we can apply the same reasoning to any collection of arc-disjoint s-t paths contained in $\hat{A}_i$.

This suggests the following heuristic: at each iteration we maintain an auxiliary network using the arcs in $\hat{A}_i$, which we henceforth call an *incremental network* and denote it by $\hat{N}_i$. At each iteration $i > 0$, we use the maximum number of arc-disjoint paths in the incremental network along with a maximum flow in N_{i-1} to construct a good feasible flow for N_i. We formally define the incremental network below.

Definition 2 The *incremental network* for iteration i is the network $\hat{N}_i = (V, \hat{A}_i)$ where all arcs have unit capacity.

Since $u^i_{e_j} - u^{i-1}_{e_j} = d_{e_{i-1}} - d_{e_i} \ \forall\ e_j \in \hat{A}_i$, we may assume without loss of generality that every arc in the incremental network has unit capacity. In light of this assumption, each incremental network has a corresponding *multiplier* λ_i where $\lambda_i = d_{e_{i-1}} - d_{e_i}$ $\forall\ i \in \{1, 2, \ldots, |A|\}$. If the incremental network has a maximum flow of $\hat{z}^{ast}_i$ units, then the network would have a maximum flow of $\lambda_i \hat{z}^{ast}_i$ units, had each arc instead been assigned a capacity of $d_{e_{i-1}} - d_{e_i}$ units.

4.4 Algorithm details

In this subsection, we provide the details of our algorithm for RobuCut. At the beginning of iteration i, we have the following information stored:

1. $N_i = (V, A_i)$, the network where we need to compute a maximum flow.
2. x^{i-1}, the maximum flow in the $(i-1)$st network $N_{i-1} = (V, A_{i-1})$.
3. $\hat{N}_i = (V, \hat{A}_i)$, the incremental network for the ith iteration along with its corresponding multiplier λ_i.
4. $\hat{x}^{i-1}$, the maximum flow in the incremental network $\hat{N}_{i-1} = (V, \hat{A}_{i-1})$.
5. A cut tripartition $\{\hat{C}^{i-1}_s, \hat{C}^{i-1}_t\}$ based on an optimal residual network of $\hat{N}_{i-1}$.
6. A cut tripartition $\{C^{i-1}_s, C^{i-1}_t\}$ based on an optimal residual network of N_{i-1}.

First we discuss how to use the maximum flow in $\hat{N}_{i-1}$ to compute the maximum flow in $\hat{N}_i$. Second, we discuss how to construct an initial feasible solution for the nominal maximum flow problem on N_i using the maximum flow in $\hat{N}_i$ and the maximum flow in N_{i-1}. Finally, we discuss computing the maximum flow in N_i.

4.4.1 Computing a maximum flow in the ith incremental network

We may assume $i > 0$ since computing a maximum flow in the 0th incremental network is trivial. Let $\{e_{i-1}\} = \hat{A}_i \backslash \hat{A}_{i-1}$. `ReoptIncNetwork(`$\hat{N}_i$`,` $\hat{x}^{i-1}$`,` e_{i-1}`,` $\{\hat{C}^{i-1}_s, \hat{C}^{i-1}_t\}$`)` is our subroutine for computing a maximum flow in $\hat{N}_i$. Algorithm 2 contains pseudocode for `ReoptIncNetwork(`$\hat{N}_i$`,` $\hat{x}^{i-1}$`,` e_{i-1}`,` $\{\hat{C}^{i-1}_s, \hat{C}^{i-1}_t\}$`)`.

`ReoptIncNetwork(`$\hat{N}_i$`,` $\hat{x}^{i-1}$`,` e_{i-1}`,` $\{\hat{C}^{i-1}_s, \hat{C}^{i-1}_t\}$`)` takes four inputs, which are all listed in the parenthesis. This subroutine returns two outputs: a maximum flow $\hat{x}^i$ and a new cut tripartition $\{\hat{C}^i_s, \hat{C}^i_t\}$.

The subroutine `findAugmentingPath(`$\hat{N}_i$`,` $\hat{x}^i$`)` finds an augmenting path in the residual network of $\hat{N}_i$ on flow $\hat{x}^i$ using depth-first search. Since incremental networks are unit capacity networks, we need to find at most one augmenting path. Lastly, the subroutine `updateCutTripartition(`$\hat{N}_i$`,` $\hat{x}^i$`,` $\{\hat{C}^{i-1}_s, \hat{C}^{i-1}_t\}$`)` takes the three inputs listed in the

$\hat{x}_e^i \leftarrow \hat{x}_e^{i-1} \; \forall \, e \in \hat{A}_{i-1}$
$\hat{x}_{e_{i-1}}^i \leftarrow 0$

if $e_{i-1} \in \hat{C}_s^{i-1} \cap \hat{C}_t^{i-1}$ **then**
  `findAugmentingPath(`$\hat{N}_i$`,` $\hat{x}^i$`)`
end if

$\{\hat{C}_s^i, \hat{C}_t^i\} \leftarrow$ `updateCutTripartition(`$\hat{N}_i$`,` $\hat{x}^i$`,` $\{\hat{C}_s^{i-1}, \hat{C}_t^{i-1}\}$`)`

return $(\hat{x}^i, \{\hat{C}_s^i, \hat{C}_t^i\})$

Algorithm 2: `ReoptIncNetwork(`$\hat{N}_i$`,` $\hat{x}^{i-1}$`,` e_{i-1}`,` $\{\hat{C}_s^{i-1}, \hat{C}_t^{i-1}\}$`)`

parenthesis and returns a new cut tripartition in the optimal residual network of $\hat{N}_i$. A cut tripartition can always be constructed from scratch by using two breadth-first search methods, one from the source and the other from the sink. However, when the maximum flow value in $\hat{N}_{i-1}$ equals the maximum flow value in $\hat{N}_i$, it is usually much faster in practice to update the cut tripartition from an optimal residual network of $\hat{N}_{i-1}$, using breadth-first search, to obtain the cut tripartition from an optimal residual network of $\hat{N}_i$.

4.4.2 Constructing a feasible flow in the ith nominal network

In this section we discuss how we construct a maximum flow in N_i given a maximum flow in an incremental network $\hat{N}_i$ and a maximum flow in the previous network N_{i-1}. To this end, we introduce the following merge operation, which is detailed in Algorithm 3.

for each e in A_i **do**
  if e is in $\hat{A}_i$ **then**
    $x_e^i \leftarrow x_e^{i-1} + \lambda_i \hat{x}_e^i$
    $u_e^i \leftarrow u_e^{i-1} + \lambda_i$
  else
    $x_e^i \leftarrow x_e^{i-1}$
    $u_e^i \leftarrow u_e^{i-1}$
  end if
end for

return x^i

Algorithm 3: `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\hat{N}_i$`,` $\hat{x}^i$`,` λ_i`)`

Algorithm 3 contains the pseudocode for the subroutine `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\hat{N}_i$`,` $\hat{x}^i$`,` λ_i`)` where the five inputs for the subroutine are contained within the parenthesis. x^{i-1} denotes a maximum flow in N_{i-1} and $\hat{x}^i$ denotes a maximum flow in $\hat{N}_i$. Thus, x_e^{i-1} and $\hat{x}_e^i$ denote the amount of flow on arc e in flows x^{i-1} and $\hat{x}^i$ respectively. `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\hat{N}_i$`,` $\hat{x}^i$`,` λ_i`)` returns x^i a feasible, but not necessarily optimal, flow in N_i.

4.4.3 Adding pre-flow to the i th nominal network

For the purpose of determining how much pre-flow to add to N_i, we obtain a quickly computable upper bound Δ on $z_i^* - z_i^{init}$ where z_i^* denotes the maximum flow value in N_i and z_i^{init} denotes the value of the flow constructed by the subroutine `MergeNetworks(`N_{i-1}, x^{i-1}, $\hat{N}_i$, $\hat{x}^i$, λ_i`)`. We know each unit of pre-flow added to N_i in excess of Δ must inevitably be returned to the source. Moreover, each unit of pre-flow in excess of Δ will most likely result in unnecessary computations. Thus, having a quickly computable upper bound on $z_i^* - z_i^{init}$ allows us to heuristically restrict the amount of pre-flow we add to N_i.

Let $(V_s^{i-1}, V\backslash(V_s^{i-1} \cup V_t^{i-1}), V_t^{i-1})$ be a cut tripartition on the optimal residual network of N_{i-1} and let $\hat{A}_i^r$ be the set of residual arcs in the optimal residual network of $\hat{N}^i$. Note that after two networks are merged, it is possible for $z_i^* - z_i^{init} > 0$, even if $\{(u,v) \in \hat{A}_i^r : u \in V_s^{i-1}, v \in V_t^{i-1}\} = \emptyset$. Nevertheless, we can compute an upper bound on $z_i^* - z_i^{init}$ using our cut tripartition $(V_s^{i-1}, V\backslash(V_s^{i-1} \cup V_t^{i-1}), V_t^{i-1})$.

Lemma 1 *The following inequality is true*:

$$(d_{e_{i-1}} - d_{e_i})\theta_i \geq z_i^* - z_i^{init} \tag{1}$$

where

$$\theta_i = \min\left\{|\{(u,v) \in \hat{A}_i^r : u \in V_s^{i-1}, v \notin V_s^{i-1}\}|,\ |\{(u,v) \in \hat{A}_i^r : v \in V_t^{i-1}, u \notin V_t^{i-1}\}|\right\}$$

Proof z_i^{init} is the objective value of the flow x^i. Moreover, given the feasible flow x^i, the maximum flow in N_i equals z_i^{init} plus the maximum flow in the residual network obtained when x^i is routed through N_i.

Let ζ^r be the set of all s-t cuts in the residual network when x^i is routed through N_i and let r_e^i be the residual capacity of arc e when flow x^i is sent through N_i. From the Maximum Flow Minimum Cut Theorem, we get:

$$z_i^* - z_i^{init} = \min_{C^r \in \zeta^r} \sum_{e \in C^r} r_e^i$$

Let A^r be the set of residual arcs when x^i is routed through N_i. Let $C_s^r = \{(u,v) \in A^r : u \in V_s^{i-1}, v \notin V_s^{i-1}\}$ and let $C_t^r = \{(u,v) \in A^r : u \notin V_t^{i-1}, v \in V_t^{i-1}\}$. Then we obtain the following inequality:

$$z_i^* - z_i^{init} \leq \min\left\{\sum_{e \in C_s^r} r_e^i, \sum_{e \in C_t^r} r_e^i\right\}$$

Since both $(V_s^{i-1}, V\backslash V_s^{i-1})$ and $(V\backslash V_t^{i-1}, V_s^{i-1})$ are minimum cuts of N_{i-1} we know $r_e^i = d_{e_{i-1}} - d_{e_i}\ \forall\, e \in C_s^r \cup C_t^r$ by construction of x^i:

$$z_i^* - z_i^{init} \leq \min\left\{\sum_{e \in C_s^r} d_{e_{i-1}} - d_{e_i}, \sum_{e \in C_t^r} d_{e_{i-1}} - d_{e_i}\right\}$$

which can be simplified to:

$$z_i^* - z_i^{init} \leq (d_{e_{i-1}} - d_{e_i}) \min\{|C_s^r|, |C_t^r|\}$$

By construction of x^i and from the topological similarities between $\hat{N}_i$ and N_i, we know $|C_s^r| = |\{(u,v) \in \hat{A}_i^r : u \in V_s^{i-1}, v \notin V_s^{i-1}\}|$ and $|C_t^r| = |\{(u,v) \in \hat{A}_i^r : v \in V_t^{i-1}, u \notin V_t^{i-1}\}|$, which completes the proof. □

4.4.4 *Computing a maximum flow in the ith nominal network*

We may now formally state our algorithm for RobuCut, whose pseudocode may be found in Algorithm 4.

$(z_0^*, x^0) \leftarrow$ `GoldbergTarjan(`N_0`)`
$\{C_s^0, C_t^0\} \leftarrow$ `constructCutTripartition(`N_0`,` x^0`)`
$\hat{x}^0 \leftarrow 0$
$\{\hat{C}_s^0, \hat{C}_t^0\} \leftarrow \{\emptyset, \emptyset\}$

for $i = 1, \ldots, |A|$ **do**

  if $e_{i-1} \in \hat{C}_s^{i-1} \cap \hat{C}_t^{i-1}$ **then**

    $(\hat{x}^i, \{\hat{C}_s^i, \hat{C}_t^i\}) \leftarrow$ `ReoptIncNetwork(`$\hat{N}_i$`,` $\hat{x}^{i-1}$`,` e_{i-1}`,` $\{\hat{C}_s^{i-1}, \hat{C}_t^{i-1}\}$`)`
  else

    $(\hat{x}^i, \{\hat{C}_s^i, \hat{C}_t^i\}) \leftarrow (\hat{x}^{i-1}, \{\hat{C}_s^{i-1}, \hat{C}_t^{i-1}\})$

  end if

  $x_{feas}^i \leftarrow$ `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\hat{N}_i$`,` $\hat{x}^i$`,` $d_{e_{i-1}} - d_{e_i}$`)`

  $\hat{\theta}_i \leftarrow$ `ComputeUB(`N_i`,` x_{feas}^i`,` $d_{e_{i-1}} - d_{e_i}$`,` $\{C_s^{i-1}, C_t^{i-1}\}$`)`

  $(z_i^*, x^i) \leftarrow$ `modMaxFlow(`N_i`,` x_{feas}^i`,` $\hat{\theta}_i$`)`

  $\{C_s^i, C_t^i\} \leftarrow$ `updateCutTripartition(`N_i`,` x^i`,` $\{C_s^{i-1}, C_t^{i-1}\}$`)`

end for

return $\min_{i \in \{0,1,\ldots,|A|\}} \Gamma d_{e_i} + z_i^*$

Algorithm 4: Algorithm for RobuCut

Algorithm 4 begins by computing a maximum flow in N_0 using the Goldberg-Tarjan algorithm along with creating a cut tripartition $\{C_s^0, C_t^0\}$ on an optimal residual network of N_0. For notational convenience, we initialize the maximum flow in the 0th incremental network $\hat{x}^0$ to be the trivial flow of 0 units and we initialize two empty cuts for the cut tripartition for the optimal residual network of the 0th incremental network $\{\hat{C}_s^0, \hat{C}_t^0\}$.

The incremental network initially starts with no arcs. Recall $e_{i-1} \in \hat{A}_i \backslash \hat{A}_{i-1}$. At the beginning of iteration i, the algorithm checks the cut tripartition corresponding to the $(i-1)$st incremental network, $\{\hat{C}_s^{i-1}, \hat{C}_t^{i-1}\}$, to see if the maximum flow from the in

the $(i-1)$st incremental network, $\hat{x}^{i-1}$, is also a maximum flow for the ith incremental network $\hat{N}_i$. If not, then we compute the new maximum flow in $\hat{N}_i$ using the subroutine `ReoptIncNetwork(`$\hat{N}_i$`,` $\hat{x}^{i-1}$`,` e_{i-1}`,` $\{\hat{C}_s^{i-1}, \hat{C}_t^{i-1}\}$`)`. Otherwise, we equate $\hat{x}^i$ and $\{\hat{C}_s^i, \hat{C}_t^i\}$ to their respective values from the previous iteration.

Given $\hat{x}^i$, we combine this with x^{i-1}, the maximum flow in the $(i-1)$st nominal network, to construct an initial feasible flow x^i_{feas} for N_i. This is done during the subroutine `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\hat{N}_i$`,` $\hat{x}^i$`,` $d_{e_{i-1}} - d_{e_i}$`)`. Next, we obtain an upper bound on the difference between the maximum flow value in N_i and the current value of flow using the subroutine `ComputeUB(`N_i`,` x^i`,` $d_{e_{i-1}} - d_{e_i}$`,` $\{C_s^{i-1}, C_t^{i-1}\}$`)`, which returns the upper bound proved in Lemma 1.

The penultimate step of an iteration is to compute the maximum flow in N_i using the subroutine `modMaxFlow(`N_i`,` x^i`,` $\hat{\theta}_i$`)`. The details of this subroutine are presented in the next paragraph. In the final step of each iteration, we update the cut tripartition for the optimal residual network of N_i using the subroutine `updateCutTripartition(`N_i`,` x^i`,` $\{C_s^{i-1}, C_t^{i-1}\}$`)`. After all iterations are complete, the algorithm returns the optimal solution to the RobuCut problem.

The subroutine `modMaxFlow(`N_i`,` x^i_{feas}`,` $\hat{\theta}_i$`)` requires three arguments, each of which are listed parenthetically, and returns two outputs. The pseudocode for this subroutine is listed in Algorithm 5.

$V \leftarrow V \cup \{\hat{s}\}$ // Create a new source $\hat{s}$.
$A \leftarrow A \cup \{(\hat{s}, s)\}$ // Add a new uncapacitated arc.
Initialize $d(v)$ $\forall$ $v \in V$ using global relabeling
$e(v) \leftarrow 0$ $\forall$ $v \in V$
$e(\hat{s}) \leftarrow \hat{\theta}_i$ $x_{(\hat{s},s)} \leftarrow \hat{\theta}_i$

while There is an active node i **do**
  if the residual network contains an admissible arc (i, j) **then**
    Push $\delta := \min\{e(i), c_{(i,j)} - x_{(i,j)}\}$ units of flow from node i to node j
  else
    $d(i) \leftarrow \min\{d(j) + 1 : (i, j) \in \text{residual } FS(i)\}$
  end if
end while

$V \leftarrow V \backslash \{\hat{s}\}$
$A \leftarrow A \backslash \{(\hat{s}, s)\}$

return $(e(t), x)$

Algorithm 5: `modMaxFlow(`N_i`,` x^i`,` $\hat{\theta}_i$`)`

`modMaxFlow(`N_i`,` x^i`,` $\hat{\theta}_i$`)` is similar to the Goldberg-Tarjan algorithm but contains two key differences. First, at most $\hat{\theta}_i$ of pre-flow is added to the arcs in $FS(s)$. This is not mandatory for correctness but instead is a heuristic improvement for reasons previously discussed.

Second, during `modMaxFlow(`N_i, x^i, $\hat{\theta}_i$`)`, a temporary new source $\hat{s}$ is added to the network and is only adjacent to the original source s. This allows the original source s to

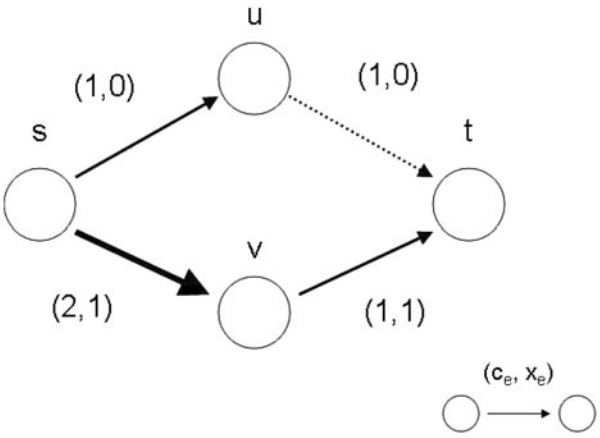

Fig. 1 Saturating the "wrong" arc

be relabeled and to potentially receive and subsequently redirect a positive excess of flow throughout the algorithm. This is in contrast to the standard Goldberg-Tarjan implementation, where the source is never relabeled and all flow that is returned to the source is immediately removed from the network.

Recall that we are only adding a bounded amount of pre-flow $\hat{\theta}_i$ to N_i. In contrast, the original Goldberg-Tarjan algorithm begins by saturating all arcs in $FS(s)$. Since we arbitrarily choose which arcs in $FS(s)$ to initially distribute $\hat{\theta}_i$ units of pre-flow, it is possible that we could saturate the "wrong" arcs. That is, at least one unit of pre-flow may be placed on an arc in $FS(s)$ where that flow cannot possibly reach the source, even if the network currently is not at maximum flow.

Figure 1 illustrates a situation where a "wrong" arc is saturated. Assume the dashed arc (u, t) has just been added to the network. In the diagram, the bold arc (s, v) has been saturated when it is better to initially saturate (s, u).

There are pathological examples where less computations are required during the course of `modMaxFlow(`N_i`,` x^i`,` $\hat{\theta}_i$`)` if all arcs in $FS(s)$ are initially saturated. However, in practice, it is typically faster to add a bounded amount of pre-flow to the network while simultaneously adding a temporary source to allow s to be relabeled.

4.5 Algorithmic result

In this subsection, we prove Algorithm 4 is no worse, in terms of worst-case complexity, than solving $|A|$ maximum flow problems from scratch using the highest label implementation of the algorithm of Goldberg and Tarjan.

Theorem 4 *Consider an instance of RobuCut on a network $N = (V, A)$. Algorithm 4 runs in time $O(|V|^2|A|^{\frac{3}{2}})$.*

Proof Cheriyan and Melhorn (1999) prove the highest label implementation of the Goldberg-Tarjan algorithm, which is used in our implementation, runs in time $O(|V|^2\sqrt{|A|})$. A modified version of this algorithm, which has the same worst-case algorithmic bound, is called $O(|A|)$ times. This leads to the bound $O(|V|^2|A|^{\frac{3}{2}})$ for all modified maximum flow computations, including solving the 0th nominal maximum flow problem.

What remains to show is a bound on solving the sequence of incremental networks. Recall that to compute the maximum flow value of the ith incremental network given a maximum flow in the $(i-1)$st incremental network as an initial solution requires the computation of at most one augmenting path. Since it takes $O(|A|)$ time to find an augmenting path in a network with $|A|$ arcs, to search for augmenting paths in the sequence of incremental networks takes $1+2+\cdots+|A| = \frac{|A|(|A|-1)}{2} \in O(|A|^2)$. Finally, since for simple networks $O(|A|) \subseteq O(|V|^2)$ this implies $O(|A|^2) \subseteq O(|V|^2|A|) \subseteq O(|V|^2|A|^{\frac{3}{2}})$. The result follows. □

5 Computational experimentation

In this section, we describe two computational experiments. The first experiment investigates the best algorithmic techniques for solving sequences of maximum flow problems that stem from RobuCuts. Specifically, we compare the performance of Algorithm 4, an algorithm that iteratively uses the Goldberg-Tarjan algorithm as a black-box subroutine (henceforth referred to as *the black-box approach*) and an algorithm that incrementally uses a maximum flow simplex algorithm (henceforth referred to as *the simplex approach*).

In the black-box approach, no warm starting techniques are used. We consider this as a valid alternative worth testing for a few reasons. First, the Goldberg-Tarjan algorithm is still considered the best algorithm for the maximum flow problem in practice (Cherkassky and Goldberg 1994; Matsuoka and Fujishige 2004). Second, computer code for intensively tested Goldberg-Tarjan solvers are widely available (Goldberg 2010). Third and most importantly, the available codes for the Goldberg-Tarjan algorithm do not have warm starting heuristics built-in.

Now we detail the simplex approach. Although the maximum flow simplex algorithm is not highly regarded for its ability to quickly solve a single maximum flow problem, simplex algorithms in general are well known for their ability to warm start efficiently after slight perturbations to problem data. Thus, since our general method for solving RobuCut consists of solving a sequence of similar maximum flow problems, this suggests that an algorithm with great potential for warm starts, such as the maximum flow simplex algorithm, is worthy of investigation.

The second experiment that we conduct demonstrates the advantage, in terms of running time, of using incremental networks to construct a better feasible solution for each maximum flow problem defined on a nominal network as opposed to merely warm starting the maximum flow problem on the ith nominal network with a maximum flow from the $(i-1)$st nominal network. This advantage of using incremental networks does not become pronounced until we solve instances of RobuCut on networks with tens of thousands of nodes. To highlight this difference, we conduct our second experiment on instances of RobuCut with substantially more nodes than those used in our first experiment.

In the first subsection, we provide greater detail on the specific implementations of the algorithms we used as well as the computer we ran our experiments on. In the second subsection, we detail how we generated test instances. In the third subsection, we describe our computational results for the experiment that compares the different algorithms. In the fourth subsection, we describe our computational results for the experiment that tests the advantage of using incremental networks.

5.1 Implementations of algorithms used

To ensure a controlled experiment, we implemented each of the tested algorithms with the same data structures, memory allocation techniques and any other considerations that might impact the running time, to the extent that it is possible. We implemented all three of our algorithms using the C programming language. Algorithm 4 is implemented using a highest-label implementation of the Goldberg-Tarjan algorithm. We implemented both the gap relabeling and the global relabeling heuristics, as described in Cherkassky and Goldberg (1994). We used the number of nodes in the network as the global relabeling frequency parameter. To implement our networks, we used the forward star representation as described in Ahuja et al. (1993).

5.1.1 The black-box approach implementation

For the black-box approach, we used our implementation of the highest-label Goldberg-Tarjan algorithm. As with Algorithm 4, we used both the gap and global relabeling heuristics as well as the forward star network representation. To reduce the running time, we only allocate memory for the network data structures at the beginning of the algorithm. Thus, when using the black-box algorithm to solve the ith maximum flow problem where $i > 0$, we simply reset all arc flows to zero and restart the distance labels on the nodes using the global relabeling heuristic.

During iteration i of the black-box algorithm, if the multiplier of the incremental network $d_{e_{i-1}} - d_{e_i}$ equals 0 then we do not recompute the maximum flow since the maximum flow in the ith nominal network equals that of the $(i-1)$st nominal network. For our generated instances, this substantially reduces the number of calls to the black box maximum flow solver.

5.1.2 The simplex approach implementation

We implemented the maximum flow simplex algorithm as described in Sect. 11.8 of Ahuja et al. (1993). To prevent the algorithm from cycling, we implemented the strongly feasible trees pivoting rule, which is described in Sect. 11.6 of Ahuja et al. (1993). We also implemented a special tree data structure to allow the dual variables on the nodes to be efficiently updated as opposed to having to relabel all of the nodes from scratch during every iteration.

Recall that a maximum flow in the ith nominal network, x^i, is always a feasible solution to $(i+1)$st nominal maximum flow problem MFP_{i+1}. However, x^i might be an interior point solution to the maximum flow problem defined on N_{i+1}, which means that x^i cannot be used as a starting basic feasible solution to MFP_{i+1}. This can only happen if we increase the capacity on one of the non-basic arcs that is at its upper bound in x^i. As a remedy, arc e is temporarily split into two parallel arcs, one that is saturated and one that has no flow. Note that by doing this, we can always obtain an initial basic feasible solution for MFP_{i+1}. Therefore, we can use this solution to warm start the maximum flow simplex algorithm.

We also note that we did not explicitly split arcs in our implementation so as not to create too many arcs in the network. Instead, we merely partitioned the non-basic arcs into three sets: non-basic arcs with zero flow, non-basic arcs that are saturated and non-basic arcs with non-zero flow and non-zero residual capacity. However, the specific implementation details are beyond the scope of this paper.

5.2 Generating test instances

To conduct our experiments, we generated three broad classes of random instances: acyclic networks, grid networks and general random (i.e., not necessarily acyclic) networks. First, we generated instances of RobuCut on both random acyclic networks and random grid networks since recent computational studies of maximum flow algorithms have been tested on such networks (Cherkassky and Goldberg 1994; Matsuoka and Fujishige 2004). We also tested the algorithms on general random networks to show our results are not limited to networks with either grid or acyclic topologies.

We first discuss how we generated instances on acyclic networks. Second, we discuss how we generated instances on networks with grid topologies. Third, we discuss how we generated instances on general random networks. Lastly, we discuss the differences between each of these network topologies in the context of computing maximum flows.

We note that both the size and the number of maximum flow computations required in Algorithm 4 is independent of Γ. Recall that our study investigates which algorithmic approach solves instances of RobuCut in the least amount of time. Thus, choosing a single value for Γ is sufficient for our experiment.

5.2.1 Acyclic network instance generation

We randomly generate instances of RobuCut on acyclic networks according to the following distribution:

1. The number of nodes is deterministically selected, as a parameter, from the set $\{200, 250, 300, 400, 450, 500\}$.
2. Given a set of nodes V, the probability that each arc in the complete acyclic network on V is included in our RobuCut instance is p, where p is a parameter that is deterministically chosen from the set $\{.55, .6, .65, .7, .75, .8, .9, .95\}$. We refer to p as the *arc density parameter*.
3. The lowest possible capacity for each arc e, u_e, is drawn uniformly at random from the interval $[10, 50]$.
4. The largest possible increase in an arc's capacity for each arc e, d_e, is drawn uniformly at random from the interval $[5, 20]$.
5. The parameter of robustness Γ is arbitrarily chosen to be 20.

For each possible value of the arc density parameter p, for each possible number of nodes, we randomly generated 10 instances. The naming convention for these instances is `acyclic-nN-p(100p).net` where N is the number of nodes. For example, the class of instances generated on 300 nodes with arc density .7 is named `acyclic-n300-d70.net`.

5.2.2 Grid topology instance generation

We randomly generate instances of RobuCut on networks with grid topologies. Figure 2 illustrates a sample grid topology with three rows and four columns of nodes. We obtain the idea for these topologies as well as Fig. 2 from Royset and Wood (2007).

1. The number of rows of nodes is deterministically selected, as a parameter, from the set $\{20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250\}$.
2. Typically, we choose the number of columns to be 2 times the number of rows. However, in our second experiment, we also generated networks where the number of columns equals $\frac{3}{2}$ and $\frac{5}{2}$ the number of rows.

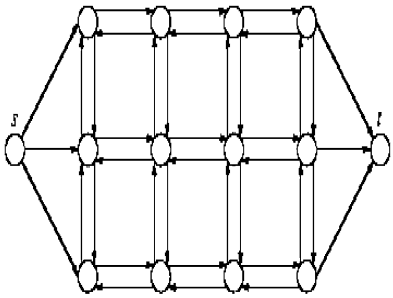

Fig. 2 Grid topology with three rows and four columns

3. The lowest possible capacity for each arc e, u_e, is drawn uniformly at random from the interval [10, 50].
4. The largest possible increase in an arc's capacity for each arc e, d_e, is drawn uniformly at random from the interval [5, 20].
5. The parameter of robustness Γ is arbitrarily chosen to be 20.

For each possible number of rows of nodes r, for each considered number of columns c, we generated 10 instances. The naming convention for these instances is `grid-rxc.net` where c is the number of columns and r is the number of rows. For example, the class of instances generated with 30 rows of nodes and 60 columns of nodes is named `grid-30x60.net`.

5.2.3 General random network instance generation

We generate instances of RobuCut on random general networks according to the following distribution:

1. The number of nodes is deterministically selected, as a parameter, from the set $\{200, 250, 300, 350, 400, 450\}$.
2. Given a set of nodes V such that $|V| \in \{200, 250, 300\}$, the probability that each arc in the complete network on V is included in our RobuCut instance is p, where p is a parameter that is deterministically chosen from the set $\{.6, .7, .8, .9\}$. We refer to p as the *arc density parameter*. For node sets such that $|V| \in \{350, 400, 450\}$, we generated arcs with p chosen from the set $\{.3, .4, .5, .6, .7, .8, .9\}$.
3. The lowest possible capacity for each arc e, u_e, is drawn uniformly at random from the interval [10, 50].
4. The largest possible increase in an arc's capacity for each arc e, d_e, is drawn uniformly at random from the interval [5, 20].
5. The parameter of robustness Γ is arbitrarily chosen to be 20.

For each possible pair of the arc density parameter p and a possible number of nodes $|V|$, we randomly generated 10 instances. The naming convention for these instances is `rand-nN-p(100p).net` where N is the number of nodes. For example, the class of instances generated on 300 nodes with arc density .7 is named `rand-n300-d70.net`.

5.2.4 Comparing the network topologies

There are two key topological differences between the grid instances and the other two instances. First, we expect the non-grid instances to be much more dense in terms of the

number of arcs. The expected degree of each node in the non-grid networks is a linear function of the number of nodes in the network. However, the degree of each node in the grid networks is bounded above by a fixed constant that does not increase with the number of nodes in the network.

Second, the shortest s-t path, in terms of the fewest number of arcs, should be much longer for the grid topologies than in the non-grid topologies. Specifically, in the grid topologies, this distance is bounded below by the number of columns of nodes. Thus, for Goldberg-Tarjan approaches that use pre-flow, there is a greater possibility of pre-flow being pushed a much greater distance towards the sink, only to be ultimately returned to the source.

The main difference between the acyclic instances and the general random instances, in the context of maximum flows, is the latter have more s-t paths, which suggests that computing the maximum flow in these networks should take longer.

5.3 Comparing the different algorithms for RobuCut

We tested the three algorithms on all of the acyclic instances and all of the general random instances. We also tested the three algorithms on all of the grid instances with up to 90 rows of nodes. Table 1 contains the results on all of the acyclic instances with less than 400 nodes. The column labeled **File Name** contains the name of the RobuCut instance class. The columns labeled **BBtime**, **MFStime**, and **AEtime** contain the average number of seconds required, over 10 randomly generated instances for the corresponding instance type, for the black-box approach, the simplex approach and Algorithm 4 to terminate with an optimal solution respectively. Note how both the black-box approach and the simplex approach require over 20 minutes for the largest of these instances while our algorithm requires around 3 seconds.

Table 2 contains the results of this experiment on all of the instances with grid topology networks that had less than 100 rows of nodes and the number of columns of nodes equals twice the number of rows of nodes. As with the smaller acyclic instances, Algorithm 4 dominates the other two approaches. Note that on the largest of these instances, the black-box algorithm requires over 25 minutes, the simplex algorithm requires over 30 minutes while our algorithm requires just over 30 seconds. As in Table 1, each of these entries is an average over 10 different instances drawn from the same distribution.

We also note that the relative performance of our algorithm, when compared to the best of the two alternate approaches, is slightly worse on the instances on grid topologies as opposed to the instances on acyclic networks. We suspect this is because our algorithm does not fully saturate the forward star of the source with flow at the beginning of each maximum flow computation. Although this heuristic typically helps under most circumstances, the low-degree of the nodes combined with the length of each s-t path bounded below by the number of columns of nodes most likely increases the chances of "wrong" arcs, as is illustrated in Fig. 1, in the forward star of the source to be saturated more frequently than with the instances on the acyclic networks.

Table 3 contains the results of this experiment on all of the instances on the acyclic networks with at least 400 nodes. Unlike Table 1 and Table 2, the cells in this table correspond to the average over 2 randomly generated instance for each instance class.

As was shown with previous two sets of instances, Algorithm 4 is clearly superior to the other two algorithms. This suggests the performance of our algorithm scales well with the

Table 1 Experiment on small acyclic instances

File name	BBtime	MFStime	AEtime
acyclic-n200-p55.net	45.3	48.4	0.3
acyclic-n200-p60.net	62.4	64.6	0.3
acyclic-n200-p65.net	81.0	85.9	0.4
acyclic-n200-p70.net	98.1	100.4	0.6
acyclic-n200-p75.net	115.0	123.0	0.7
acyclic-n200-p80.net	143.2	146.3	0.9
acyclic-n250-p55.net	145.2	151.4	0.5
acyclic-n200-p85.net	160.7	169.8	0.8
acyclic-n200-p90.net	199.6	202.3	0.6
acyclic-n250-p60.net	201.5	201.6	0.8
acyclic-n200-p95.net	246.7	261.6	1.1
acyclic-n250-p65.net	253.0	263.5	1.1
acyclic-n250-p70.net	305.6	309.4	1.2
acyclic-n250-p75.net	331.0	343.9	1.4
acyclic-n300-p55.net	395.8	408.4	1.2
acyclic-n250-p80.net	424.6	434.0	1.4
acyclic-n300-p60.net	468.1	483.8	1.4
acyclic-n250-p85.net	516.3	538.3	1.4
acyclic-n250-p90.net	587.8	597.9	1.9
acyclic-n300-p65.net	600.9	621.6	1.9
acyclic-n250-p95.net	694.4	725.1	1.9
acyclic-n300-p75.net	888.4	917.7	2.4
acyclic-n300-p85.net	1288.4	1322.9	3.1

Table 2 Experiment on small grid instances

File name	BBtime	MFStime	AEtime
grid-20x40.net	1.0	0.6	0.3
grid-30x60.net	6.2	8.1	0.9
grid-40x80.net	34.0	41.9	1.3
grid-50x100.net	132.5	152.2	3.6
grid-60x120.net	239.9	290.8	7.3
grid-70x140.net	530.5	635.7	12.1
grid-80x160.net	939.0	1141.9	21.3
grid-90x180.net	1556.8	1901.8	36.8

number of nodes in a RobuCut instance. Note that on the largest of these instances, both the black-box algorithm and the simplex algorithm require over 4.5 hours on average while our algorithm requires a little under 15 seconds on average.

Table 4 contains the results of this experiment on all of the instances on the general random networks with 200, 250 or 300 nodes but only contains the instances with 350, 400 or 450 nodes that have an arc density parameter selected from the set {.3, .4, .5}. The

Table 3 Results on large acyclic instances

File name	BBtime	MFStime	AEtime
acyclic-n400-p60.net	1855.0	1896.5	4.0
acyclic-n400-p70.net	2809.0	2850.5	4.5
acyclic-n450-p60.net	3233.0	3313.5	5.0
acyclic-n400-p80.net	4041.5	4116.5	6.5
acyclic-n450-p70.net	4850.0	4942.5	7.0
acyclic-n500-p60.net	5368.5	5457.0	7.0
acyclic-n400-p90.net	5561.0	5650.0	7.0
acyclic-n450-p80.net	7069.0	7171.0	9.0
acyclic-n500-p70.net	8156.0	8310.0	10.0
acyclic-n450-p90.net	9751.0	9878.5	11.5
acyclic-n500-p80.net	11878.0	12028.0	11.5
acyclic-n500-p90.net	16325.0	16512.0	14.5

Table 4 Results on general random instances

File name	BBtime	MFStime	AEtime
rand-n200-p60.net	353.6	354.3	1.3
rand-n200-p70.net	539.1	539.9	1.5
rand-n200-p80.net	783.9	781.9	2.6
rand-n200-p90.net	1051.9	1046.9	2.9
rand-n250-p60.net	1041.8	1038.9	2.7
rand-n250-p70.net	1573.8	1571.7	3.7
rand-n250-p80.net	2272.3	2249.9	4.9
rand-n250-p90.net	3120.9	3087.8	6.0
rand-n300-p60.net	2469.4	2445.5	4.8
rand-n300-p70.net	3709.0	3671.5	6.6
rand-n300-p80.net	5358.4	5332.6	8.6
rand-n300-p90.net	7391.4	7326.7	10.9
rand-n350-p30.net	803.4	795.7	2.3
rand-n350-p40.net	1694.7	1695.4	3.7
rand-n350-p50.net	3091.7	3065.8	5.5
rand-n400-p30.net	1537.5	1519.8	3.3
rand-n400-p40.net	3269.9	3238.1	5.8
rand-n400-p50.net	5893.6	5852.8	8.5
rand-n450-p30.net	2606.8	2599.2	4.8
rand-n450-p40.net	5627.6	5601.6	8.0
rand-n450-p50.net	10269.2	10201.0	12.0

remaining general random instances are contained in Table 5 and are discussed separately. The cells in Table 4 correspond to the average of 10 randomly generated instances for each instance class.

Table 5 Results on large dense general random instances

File name	AEtime
rand-n350-p60.net	8.8
rand-n350-p70.net	11.8
rand-n350-p80.net	14.7
rand-n350-p90.net	18.4
rand-n400-p60.net	12.7
rand-n400-p70.net	17.0
rand-n400-p80.net	21.5
rand-n400-p90.net	26.6
rand-n450-p60.net	18.3
rand-n450-p70.net	24.0
rand-n450-p80.net	30.9
rand-n450-p90.net	38.7

As with all previous results, Algorithm 4 is clearly superior to both the black-box approach and the simplex approach. This shows the performance of our algorithm is not dependent on the fact that the network topology is either a grid or an acyclic directed graph.

Table 5 contains the results of Algorithm 4 on all of the instances on general random networks with either 350, 400 or 450 nodes that have an arc density parameter from the set {.6, .7, .8, .9}. The cells in this table correspond to the average of 10 randomly generated instances for each instance class.

As with all previous results, Algorithm 4 still solves these instances on average in less than one minute. Both the black-box approach and the simplex approach require several hours to solve each one of these larger and denser general random instances so they were excluded from this experiment. Nevertheless, these results demonstrate that our algorithm performs well even on instances on large dense general random networks.

5.4 Testing the effectiveness of incremental networks

We also conducted an experiment to demonstrate the advantage of using incremental networks to heuristically improve the running time of our algorithm. To this end, we implemented an algorithm for RobuCut that is essentially Algorithm 4 without incremental networks. Thus, in this alternate algorithm, the maximum flow in jth nominal network is computed by using our implementation of the Goldberg-Tarjan algorithm with the bounded pre-flow that uses the maximum flow in the $(j-1)$st nominal network as an initial solution.

For most RobuCut instances on networks on the order of a few hundred nodes, the difference between our algorithm and our algorithm without using incremental networks was not very noticeable. Thus, we limited our comparison of these two algorithms on grid topology networks with at least 100 rows of nodes.

Table 6 contains the results of this experiment. The column labeled **File Name** contains the name of the class of RobuCut instances. The columns **AEtime** and **noINCtime** contain the average number of seconds to solve each of the 10 RobuCut instances for our algorithm and the version of our algorithm without using incremental networks respectively. The column **PerAdv?** contains the percentage of instances for this class where the algorithm using incremental networks is faster.

These results suggest using incremental networks is superior. Moreover, these results suggest the savings that stem from using the incremental networks increases along with the

Table 6 Advantage of using incremental networks

File name	AEtime	noINCtime	PerAdv?
grid-100x150.net	31.6	32.5	90%
grid-100x200.net	50.4	54.8	100%
grid-100x250.net	82.5	88.0	90%
grid-150x225.net	147.4	162.9	90%
grid-150x300.net	265.8	276.4	70%
grid-150x375.net	337.0	392.4	90%
grid-200x400.net	765.7	833.1	90%

size of the RobuCut instances. This second result is intuitive, since if an s-t path in the incremental is discovered during iteration i, it not only allows one to construct an initial feasible solution for the maximum flow problem on the ith nominal network with a greater objective value than we could without the incremental network, it also allows us to do the same for each iteration j for all $j > i$. Thus, as the size of the RobuCut instance gets larger, there are more arcs and therefore more iterations are required for our algorithm. Thus, there is greater potential for savings from using incremental networks.

6 Conclusions

In this paper, we introduce, motivate and develop an algorithm for the Robust Minimum Capacity s-t Cut Problem (RobuCut) that exploits powerful maximum flow reoptimization heuristics. RobuCut is an interesting case study in maximum flow reoptimization since the sequence of maximum flow problems that arise possesses a structure that can be exploited for problem-specific heuristics, such as the usage of incremental networks, in addition to general maximum flow reoptimization heuristics.

We have proven our algorithm is theoretically efficient. Specifically, we prove our algorithm is no worse, in terms of worst-case complexity, than solving a linear number of maximum flow problems using the highest-label Goldberg-Tarjan algorithm. This is what the worst-case complexity bound is if the Bertsimas-Sim algorithm for general RobuCOPs is directly applied to RobuCut while iteratively using the highest-label Goldberg-Tarjan algorithm as a black-box subroutine.

Furthermore, we demonstrate our algorithm is very efficient in practice. Our experiments demonstrate the substantial computational savings of maximum flow reoptimization in the context of computing RobuCuts. In particular, our algorithm can solve the largest instances tested on the order of seconds, while the alternate approaches considered in this paper can take several hours.

Since RobuCut naturally extends many industrial applications of minimum cuts, we are confident this research can be incorporated into decision tools for industries where the need to make conservative decisions in light of uncertain data is of great importance. In particular, we identify open-pit mining as an auspicious opportunity for our model. Furthermore, our algorithm has the obvious advantage in that it can solve instances of RobuCut on tens of thousands of nodes in seconds. Another advantage is that the maximum flow value in each nominal network is completely independent of the robust parameter of optimization. Thus, if a project manager wants to resolve a RobuCut instance using a different degree of conservative planning, the software can resolve without any further maximum flow computations.

We hope this paper convinces the reader that not only is it possible to save a substantial amount of computational time by using maximum flow reoptimization heuristics but that these savings can be obtained on important problems. We also hope this paper continues interest in the budding field of robust network optimization.

Acknowledgement Özlem Ergun was partially supported by the NSF CAREER Grant DMI-0238815.

References

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network flows: theory, algorithms and applications*. New York: Prentice Hall.

Altner, D. S. (2008). Advancements on problems involving maximum flows. Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia.

Altner, D. S., & Ergun, Ö. (2008). Rapidly solving an online sequence of maximum flow problems with applications to computing robust minimum cuts. In L. Perron & M. A. Trick (Eds.), *Lecture Notes in Computer Science*: *Vol. 5015. Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*. Berlin: Springer.

Atamtürk, A., & Zhang, M. (2007). Two-stage robust network flow and design under demand uncertainty. *Operations Research*, *55*, 662–673.

Ben-Tal, A., & Nemirovski, A. (1998). Robust convex optimization. *Mathematics of Operations Research*, *23*, 769–805.

Bertsimas, D., & Sim, M. (2003). Robust discrete optimization and network flows. *Mathematical Programming*, *98*, 49–71.

Chaerani, D., & Roos, C. (2006). Modelling some robust design problems via conic optimization. *Operations Research Proceedings*, 209–214.

Cheriyan, J., & Melhorn, K. (1999). An analysis of the highest-level selection rule in the preflow-push max-flow algorithm. *Information Processing Letters*, *69*, 239–242.

Cherkassky, B. V., & Goldberg, A. V. (1994). On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, *19*, 390–410.

Ford, L. R., & Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics*, *8*, 399–404.

Goldberg, A. V. (2010). Andrew V. Goldberg's network optimization library. http://avglab.com/andrew/soft.html.

Goldberg, A. V., & Rao, S. (1998). Beyond the flow decomposition barrier. *Journal of Associated Computing Machinery*, *45*, 783–797.

Goldberg, A. V., & Tarjan, R. E. (1988). A new approach to the maximum flow problem. *Journal of Associated Computing Machinery*, *35*, 921–940.

Govindaraju, V. (2008). Professor of computer science and engineering, University of Buffalo, personal communication.

Harris, T. E., & Ross, F. S. (1955). Fundamentals of a method for evaluating rail network capacities. Research Memorandum RM-1573, The RAND Corporation, Santa Monica, CA.

Hochbaum, D. S., & Chen, A. (2000). Improved planning for the open-pit mining problem. *Operations Research*, *48*, 894–914.

Matsuoka, Y., & Fujishige, S. (2004). Practical efficiency of maximum flow algorithms using maximum adjacency (MA) orderings. Technical Report METR 2004-27, University of Tokyo.

Möhring, R. H., Schulz, A. S., Stork, F., & Uetz, M. (2003). Solving project scheduling problems by minimum cut computations. *Management Science*, *49*, 330–350.

Monkhouse, P. H. L., & Yeates, G. (2005). Beyond naive optimization. *AUSIMM Spectrum Series*, *14*, 3–8.

Ordóñez, F., & Zhao, J. (2007). Robust capacity expansion of network flows. *Networks*, *50*, 136–145.

Régin, J. C. (1994). A filtering algorithm for constraints of difference in constraint satisfaction problems. In *Proceedings of the twelfth national conference on artificial intelligence*, vol. 1, pp. 362–367.

Royset, J. O., & Wood, R. K. (2007). Solving the bi-objective maximum flow network interdiction problem. *INFORMS Journal on Computing*, *19*, 175–184.

Stone, H. S. (1977). Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, *3*, 85–93.

Strickland, D. M., Barnes, E., & Sokol, J. S. (2005). Optimal protein structure alignment using maximum cliques. *Operations Research*, *53*, 389–402.